Optimal Packing in Simple-Family Codecs

Andrew Trotman
eBay Inc.
San Jose, USA

Michael Albert
Department of Computer Science
University of Otago
Dunedin, New Zealand

Blake Burgess
Department of Computer Science
University of Otago
Dunedin, New Zealand

ABSTRACT

The Simple family of codecs is popular for encoding postings lists for a search engine because they are both space effective and time efficient at decoding. These algorithms pack as many integers into a codeword as possible before moving on to the next codeword. This technique is known as left-greedy. This contribution proves that left-greedy is not optimal and then goes on to introduce a dynamic programming solution to find the optimal packing. Experiments on .gov2 and INEX Wikipedia 2009 show that although this is an interesting theoretical result, left-greedy is empirically near optimal in effectiveness and efficiency.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing - *Indexing methods*

General Terms

Algorithms, Performance.

Keywords

Inverted Files, Compression, Procrastination.

1. INTRODUCTION

The typical index seen in a search engine is known as an inverted file. An inverted file stores a vocabulary of all unique terms seen in the collection along with a postings list for each term. These postings lists are usually represented as an ordered list of <*d*, *tf*> tuples, where *d* is a document identifier and *tf* is the term frequency (more accurately, the number of times the term occurs in the document). As indexing can be performed in a single linear pass over the document collection, the document identifiers form a strictly monotonically increasing sequence but the term frequencies do not.

In a term-frequency ordered index [9] the <*d*, *tf*> tuples are sorted first on decreasing *tf*, then on increasing *d*. The index is thus represented $<tf_1{:}d_{1,1},\ d_{1,2},\ \ldots\ d_{1,n}>\ldots<tf_m{:}\ d_{m,1},\ d_{m,2},\ d_{m,n}>$ where *tf* scores decrease as *m* increases and *d* scores increase as *n* increases. The sequences of document identifiers continue to form monotonically increasing sequences; however such a representation is smaller than a document-ordered index as fewer integers are stored overall.

In a process known as *impact ordering* [2], the ranking function is partially computed at indexing time and the result is quantized into a fixed sized value (typically a single-byte or smaller). In this case the $<tf_m{:}\ d_{m,1},\ d_{m,2},\ d_{m,n}>$ sequences in the term-frequency ordered index are replaced with $<q_m{:}\ d_{m,1},\ d_{m,2},\ d_{m,n}>$ sequences, where q_m is the quantized impact of the term with respect to the document. Such an impact score might be computed from a ranking function such as BM25 [10]. Regardless of how it is computed, the document identifiers continue to form strictly monotonically increasing sequences. An impact ordered index is typically larger than a term-frequency ordered index, but faster to process.

Postings lists are normally compressed in order to reduce their size and to increase throughput. A substantial amount of prior work exists on this topic.

First a monotonic sequence is converted into a series of d-gaps [8] (also known as deltas, or differences). There are two popular approaches. In the first, known as D1 and used herein, each d-gap, g_n, is computed by subtracting the previous integer, d_{n-1} from the current integer, d_n: $g_n = d_n - d_{n-1}$. For example, the sequence <3, 5, 8, 21, 23, 24, 26, 28> becomes <3, 2, 3, 13, 2, 1, 2, 2>. These d-gap sequences further compress more effectively than when d-gaps are not used because each g_n can be no larger than d_n.

In the second approach, known as D4, four such interleaved d-gap sequences are constructed; $(g_n, g_{n+1}, g_{n+2}, g_{n+3}) = (d_n, d_{n+1}, d_{n+2}, d_{n+3}) - (d_{n-4}, d_{n-3}, d_{n-2}, d_{n-1})$. In this way the sequence <3, 5, 8, 21, 23, 24, 26, 28> becomes <3, 5, 8, 21, 20, 19, 18, 7>. This second approach is seen with schemes that use SIMD instructions to decode [8].

There are four approaches to compressing these d-gaps. The first, bit-aligned codes, is typified by schemes such as Elias gamma [5] and Golomb [6] encoding. The second, byte aligned codes, is typified by Variable Byte Encoding [11] and Group Varint [4]. The third, word-aligned codes (also known as the Simple family), is typified by Simple-9 [1], Simple-16 [15], Simple-8b [3], and variants such as PForDelta [16] and VSEncoding [12]. The fourth are SIMD schemes such as SIMD-BP128 [7] and QMX [13].

Common to the third and fourth approaches is the task of packing integers into machine words. This is typically implemented in a left-greedy fashion, packing as many integers as possible into the current codeword before moving on to the next. We ask:

Is left-greedy packing optimal?

And show by counter example that it is not.

We then present a graph-based model of the optimal packing, and a dynamic programming solution to find it.

We apply it to three members of the Simple family resulting in *Simple-9 packed*, *Simple-16 packed*, and *Simple-8b packed*.

Experiments on two standard collections show a small but negligible difference in both space effectiveness and decoding efficiency. Despite the elegance of being optimal, empirically we find that: *left-greedy packing is near optimal in space and decoding time*.

2. SIMPLE ENCODING

This section provides an overview of three Simple family codecs and discusses left-greedy packing.

ICTIR'15, September 27-30, 2015, Northampton, MA, USA
ISBN 978-1-4503-3833-2/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2808194.2809483

2.1 Simple-9

In Simple-9 [1] a 32-bit codeword is broken into two parts, (called snips), a *selector* and a *payload*. The payload carries as many fixed-width integers as possible, while the selector gives the number of integers in the payload (and hence their width in bits). For Simple-9 the selector is 4 bits wide and the payload is 28 bits wide.

Table 1: Simple-9 integer packings

Selector	0	1	2	3	4	5	6	7	8
Integers	1	2	3	4	5	7	9	14	28
Width	28	14	9	7	5	4	3	2	1

Table 1 lists the 9 possible ways to pack fixed-width integers into a 28-bit payload. The first row lists the selectors, the second lists the number of integers, while the third row lists the width of each integer. For example, encoding the sequence <260, 270, 240> using Simple-9 requires 9 bits per integer, and hence the sequence would become (in binary): <0010><100000100, 100001110, 011110000>.

2.2 Simple-16

Simple-16 [15] adds two space savings to Simple-9. First, it is observed that only 9 of the possible 16 selectors are used; in Simple-16 the remaining 7 are used to store asymmetric combinations. The second saving is that some Simple-9 combinations result in wasted bits; for example, storing five 5-bit integers leaves three unused bit. Simple-16 alters these selectors to make them asymmetric and in doing so is no worse than the Simple-9 original.

Table 2 lists the way the selectors are used in Simple-16. The first column gives the selector value, the second gives the number of integers that are being stored, while the third gives the usage of the 28-bits (the remaining columns are discussed in Section 2.3). For example, the selector value 6 represents one 3-bit integer followed by four 4-bit integers followed by three 3-bit integers.

The first space saving is seen in a selector such as 2, which stores 21 integers in a combination not possible in Simple-9. The second space saving is seen in selectors such as 10 and 11 which store combinations of 5-bit and 6-bit integers without wastage whereas Simple 9 stores five 5-bit integers with 3 bits wasted.

2.3 Simple-8b

Both Simple-9 and Simple-16 store codewords in 32-bit integers made up of selectors and payloads. Simple-8b [3] extends the size of the codeword to 64-bits, but retains the 4-bit selector. Doing so results in a space saving because fewer selectors are stored per encoded bit (4 per 60 encoded bits rather than 4 per 28 encoded bits).

Table 2 additionally lists the Simple-8b selectors and their meanings. The first column gives the selector value, the fourth column the number of integers being stored and the fifth gives the width in bits. For example, selector 6 indicates that the payload stores twelve 5-bit integers.

The space saving comes when a long sequence of similar sized integers must be stored, for example twelve 5-bit integers can be stored in one 64-bit codeword with Simple-8b whereas it would take three 32-bit codewords using Simple-9. A second space saving comes due to the addition of selectors 0 and 1 which encode long sequences of 0s.

When decoded on a 64-bit architecture Simple-8b is more efficient than Simple-9 because 64-bit instructions can be used to perform shifts during decoding.

Table 2: Simple-16 and Simple-8b integer packings

	Simple-16		**Simple-8b**	
Selector	**Integers**	**Integers × Width**	**Integers**	**Width**
0	28	28×1	240	0
1	21	7×2, 14×1	120	0
2	21	7×1, 7×2, 7×1	60	1
3	21	14×1, 7×2	30	2
4	14	14×2	20	3
5	9	1×4, 8×3	15	4
6	8	1×3, 4×4, 3×3	12	5
7	7	7×4	10	6
8	6	4×5, 2×4	8	7
9	6	2×4, 4×5	7	8
10	5	3×6, 2×5	6	10
11	5	2×5, 3×6	5	12
12	4	4×7	4	15
13	3	1×10, 2×9	3	20
14	2	2×14	2	30
15	1	1×28	1	60

2.4 Encoding

Encoding integers using the Simple family is straightforward. Compute the number of bits necessary to store the first integer in the sequence. If storing that number of bits fills a codeword then move on to the next codeword. If not then examine the next integer, and so on until a codeword is full, and then move on to the next codeword.

This packing approach is left-greedy. As many integers as possible are packed into the current codeword before moving on to the next.

3. PACKING

This section shows that left-greedy is not optimal and then introduces the optimal packing algorithm.

3.1 Non-optimal Left-greedy Packing

Left-greedy can be shown to be non-optimal by using a proof by counter example using Simple-9

Take the thirty-two integer sequence <260, 260, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 260, 260>, that is, two 9-bit integers followed by twenty-eight 1-bit integers, followed by two 9-bit integers. Such a sequence might be the d-gaps for a single term in the inverted index.

Packing left-greedy, three 9-bit integers <260, 260, 1> are packed, then fourteen 2-bit integers <1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1>, then seven 4-bit integers <1, 1, 1, 1, 1, 1, 1> then five 5-bit integers <1, 1, 1, 1, 1> then three 9-bit integers <1, 260, 260> for a total of 5 codewords.

A smaller packing stores two 14-bit integers <260, 260> then twenty-eight 1-bit integers <1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1> then two 14-bit integers <260, 260> for a total of 3 codewords.

Hence, by counter example, left-packing is not space-optimal for all sequences.

3.2 Optimal Packing

In Simple-9 there are at most 9 possible ways any one integer might be packed (see Table 1). As such, an integer sequence can be thought of as a 9-way branching tree with each vertex representing the start of the next codeword and each edge labeled with the number of integers to pack to get to that codeword. The optimal packing is the shortest valid path through this tree, the path that touches the minimum number of nodes (ignoring edge weights). A similar construction exists for Simple-16, Simple-8b, and similar codecs. Without loss of generality, the solution to Simple-9 is presented.

For long integer sequences, such as long postings lists, it is prohibitively expensive to build and exhaustively search the tree. The optimal path can, however, be built using dynamic programming. Doing so requires starting at the final integer to be encoded, the right hand end of the sequence, and working backwards. At each step the optimal solution *up to that point* is computed.

The first integer can be optimally packed into one codeword, so store a 1 for this integer (in extra storage). Moving back one integer, there are at most two ways the integer might be packed: on its own, or with the next integer. Compute all valid packings and store for this integer: the minimum number of codewords necessary to store this integer and the tail of the sequence, and which branch in the tree to take to get that packing. Move to the previous integer, which might be packed in three ways, and repeat the process. By the tenth integer there are at most nine combinations to check – one of which is optimal if the sequence started there; so store the optimal number of codewords, and the branch to take. Repeat the process until the start of the list is reached.

At each point the extra storage holds the optimal number of codewords necessary to store that integer and the tail of the sequence, along with the optimal branch to take if starting at that point. At most 9 branches need be followed for each integer – and hence the computation is linear time in the length of the sequence to be compressed, O(n).

Given the extra storage it is possible to now move forward and pack. Start at the left-hand end of the extra storage. Repeatedly follow the tree-branch and packing accordingly. As each branch is optimal starting at that point, so to must be the result. The packing process is also linear time, O(n).

As stated at the start of this section, this approach is optimal. At all times the optimal packing of the tail of the list is known, and the optimal way to add one integer to the start is computed.

4. EXPERIMENTS

Two experiments were conducted, one measuring the effect, per integer, on the index, the other measuring the effect *in situ* in the search engine.

4.1 Effect Per Integer

The two packing approaches discussed in Section 3 (left-greedy and optimal packing) were implemented for the 3 Simple family schemes discussed in Section 2 giving a total of 6 schemes. Implementation was in the ATIRE search engine [14].

For the experiments the left-greedy algorithms are known as S9, S16, S8b for Simple-9, Simple-16 and Simple-8b. The optimally packed versions are suffixed with a ‘p’ (e.g. S9p).

Experiments were conducted on two collections. First, the TREC .gov2 collection of 25,205,179 web pages crawled from the .gov domain in 2004. Second, the INEX 2009 Wikipedia collection of 2,666,190 documents, a dump taken in 2008 annotated using YAGO and converted into XML. A single core of a 64-core AMD Opteron 6276 at 2.3GHz with 512 GB RAM was used throughout.

A term-frequency ordered index was built for each collection and the mean number of bits per integer used to store the document identifiers was recorded. The term-frequency index was preferred because it is smaller than a document-ordered or impact-ordered index of the same collection. The time taken to decode each postings list was measured 10 times using the `CPUID RDTSC` combination and the minimum was recorded. The minimum of 10 was preferred because it reduces the effect of outliers.

Table 3 presents the size and time required to store and decode each document identifier in the postings lists (7,688,185,119 for .gov2 and 813,998,392 for Wikipedia). The first column gives the name of the codec, the second column the average size of an integer in bits per integer (bpi), the third column the mean time to decompress in cycles per integer (cpi). The table shows that, as expected, a Simple-8b index is smaller than a Simple-16 index, which is smaller than a Simple-9 index. It also shows virtually no difference between the left-greedy and optimal versions of each codec – suggesting that left-greedy is near-optimal.

Table 3: bits per integer (bpi) and cycles per integer (cpi) averaged over the entire index

	.gov2		**Wikipedia**	
	bpi	**cpi**	**bpi**	**cpi**
S9	11.09	6.87	11.85	9.29
S9p	11.08	6.87	11.84	9.31
S16	10.73	6.71	11.49	9.01
S16p	10.73	6.72	11.49	9.02
S8b	10.00	4.56	11.01	6.79
S8bp	9.99	4.55	11.00	6.77

4.2 Effect *In Situ*

A search engine index contains many short postings list, which are unlikely to appear in queries; it also contains many long lists also unlikely to appear in queries. This section measures the index size, and time effect on the search engine by using queries.

The size of the six indexes (in gigabytes) is presented in Figure 1. Top shows .gov2 and bottom shows Wikipedia. ATIRE produces only a single index file containing postings, vocab, external docids, and so on; so these figures show the true size of the entire index not just the postings size. The figure also shows that there is little effect due to optimal-packing.

To measure the throughput effect on the Wikipedia collection, the titles of all the assessed topics available for it (2009 & 2010, 120 topics in total) were used. For .gov2, the titles of the TREC *ad hoc* topics 701-850 were used. The search process was annotated to measure the time to fully decompress the postings lists before being processed. The experiment was repeated 95 times (overnight). The mean time to decode all lists is presented as it is reasonably resembles the expected time, however variation was very small, with standard deviations always less than 1% of the mean.

Figure 2 presents the sum of times required to decode the lists. For example, it took approximately 2 seconds to decode all the postings lists for the 150 topics for .gov2. The figure shows virtually no difference between Simple-9 and Simple-16, but Simple-8b is more efficient. The variation between the left-greedy and optimal packing is negligible, varying from 2% worse to 2% better; simple-8b, however, took 36% less time than Simple-9. The improvement due to switching schemes is vastly greater than due to the packing strategy – again suggesting that left-greedy is near optimal.

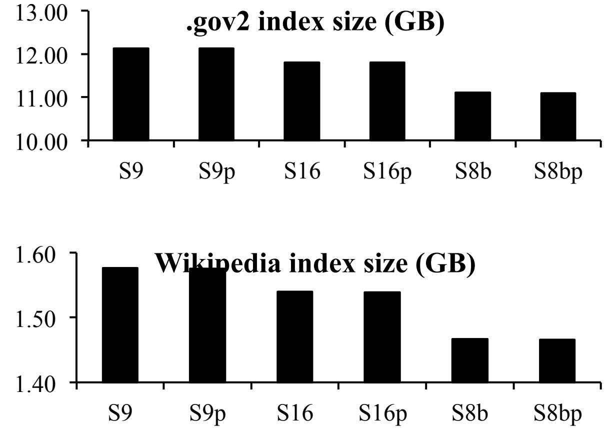

Figure 1: Index size in GB

5. CONCLUSIONS

In an inverted file based search engine the postings lists are typically compressed using d-gaps and then further compressed using a scheme such as Simple-9, Simple-16, or Simple-8b.

Compression is used for two reasons. First it can reduce the size of the index and second it can increase throughput. If the index is stored in memory, as is often the case, then the space saving makes it possible to store the index of a larger number of documents in the same amount of space. If the index is stored on disk then the reduction in size decreases the time necessary to read a postings list from disk. Regardless of where the index is stored, touching fewer memory cells to achieve the same goal can decrease processing time in a system that is memory bandwidth limited (such as a modern PC). The Simple family of compression algorithms has proven popular because schemes such as Simple-9 are both space efficient and fast to decode whereas previous schemes such as Elias gamma and Golomb were space efficient but costly to decode.

This investigation proved by counter example that the left-greedy approach to packing integers into codewords typically seen in implementations of the Simple family is not optimal. The optimal packing is given as the shortest path through the tree representing all possible packings. A linear time dynamic programming algorithm is given for computing this.

Experiments conducted on .gov2 and the INEX Wikipedia 2009 collections compared left-greedy with optimal and show negligible difference between the two. This suggests that the prior use of left-greedy has been effective and should be continued. Despite not being optimal, left-greedy is straightforward to implement and requires less work to compute.

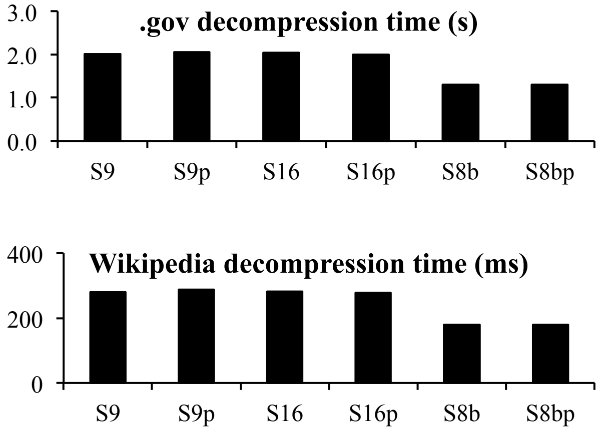

Figure 2: Mean time to decompress

REFERENCES

[1] Anh, V.N., A. Moffat, Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151-166, 2005.

[2] Anh, V.N., A. Moffat, Simplified Similarity Scoring Using Term Ranks. In *SIGIR 2005*, pp. 226-233

[3] Anh, V.N., A. Moffat, Index Compression Using 64-bit words. *Software Practice & Experience*, 40(2):131-147, 2010.

[4] Dean, J. Challenges in Building Large-Scale Information Retrieval Systems. In *WSDM 2009*

[5] Elias, P., Universal Codeword Sets and the Representation of the Integers. *IEEE Transactions on Information Theory*, 21(2):194-203, 1975.

[6] Golomb, S.W., Run-length Encodings. *IEEE Transactions on Information Theory*, 12(3):399-401, 1966.

[7] Lemire, D., L. Boytsov, Decoding Billions of Integers per Second Through Vectorization. *Software: Practice & Experience*, To Appear.

[8] Lemire, D., L. Boytsov, N. Kurz, SIMD Compression and the Intersection of Sorted Integers. *CoRR abs/1401.6399*, 2014.

[9] Persin, M., J. Zobel, R. Sacks-Davis, Filtered Document retrieval with Frequency-Sorted Indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749-764, 1996.

[10] Robertson, S.E., S. Walker, S. Jones, M.M. Beaulieu, M. Gatford. Okapi at TREC-3. In TREC-3, pp. 109-126, 1994.

[11] Scholer, F., H.E. Williams, J. Yiannis, J. Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *SIGIR 2002*, pp. 222-229

[12] Silvestri, F., R. Venturini, VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming, In *CIKM 2010*, 1219-1228.

[13] Trotman, A., Compression, SIMD, and Postings Lists, In ADCS 2014, pp. 50-57.

[14] Trotman, A., X. Jia, M. Crane, Towards an Efficient and Effective Search Engine, In *SIGIR 2012 Workshop on Open Source Information Retrieval*. pp. 40-47.

[15] Zhang, J., X. Long, T. Suel, Performance of Compressed Inverted List Caching in Search Engines, In *WWW 2008*, pp. 387-396.

[16] Zukowski, M., S. Heman, N. Nes, P. Boncz, Super-Scalar RAM-CPU Cache Compression, In *ICDE 2006*.